Organisme(s) : *EDF-R&D/AMA*

Manuel d'Utilisation
Fascicule U1.0- : Introduction à *Code_Aster*
Document : U1.03.01

Superviseur et langage de commande

Résumé :

Ce document décrit le rôle et le fonctionnement :

- du superviseur qui assure le pilotage d'une exécution de *Code_Aster* ;
- et du langage de commande qui assure la communication entre l'utilisateur et le Code.

Table des matières

1 Introduction

Le rôle du superviseur est d'assurer le commandement du déroulement d'opérations en cours d'exécution d'un programme. Les instructions d'exécution sont généralement fournies par l'utilisateur. Ceci nécessite une formalisation des communications entre le code et son exploitant, c'est le *langage de commande*.

Le langage Python est employé pour écrire le catalogue de commandes, le superviseur et les fichiers de commandes utilisateur. Pour les fichiers de commande, cela permet de décharger le superviseur de la tâche d'analyse syntaxique, dévolue à Python lui-même.

Un fichier de commandes est une succession d'appel à des fonctions Python (les commandes), définies dans le catalogue de commandes. Ces fonctions possèdent des arguments d'entrée : les mots clés et leur contenu, et des arguments de sortie : les concepts produits. L'utilisateur qui compose son fichier de commandes doit donc se soumettre à la syntaxe générale de Python (parenthèsage, indentation …) et aux règles imposées par le catalogue de commandes (les arguments fournis sont cohérents avec ce qu'attend la fonction).

Pour une première prise de contact avec le code, le lecteur pourra ne pas aborder le chapitre 2.

2 Mécanisme général de fonctionnement du superviseur

2.1 Architecture générale

Les éléments de base mis en jeu lors d'une exécution d'un calcul *Aster* sont :

- le fichier de commandes, fourni par l'utilisateur,
- le catalogue de commandes : c'est un module python de nom `cata` placé dans le package `Cata`,
- l'objet de haut niveau SUPERVISEUR,
- l'objet JDC créé par ce dernier et qui est finalement exécuté.

L'objet SUPERVISEUR est un objet python qui analyse les options transmises sur la ligne de commande, importe le catalogue de commandes, créé l'objet JDC à partir du fichier de commandes et exécute celui-ci.

L'objet JDC (nom pour Jeu De Commandes) est un objet python créé par l'objet SUPERVISEUR à partir du texte du fichier de commandes et du module catalogue de commandes. Il contient les objets ETAPE. L'objet JDC est représentatif du fichier de commandes utilisateur.

Les objets ETAPE sont représentatifs de chacun des appels à des commandes Aster dans le fichier de commande. Chaque objet ETAPE porte le nom de la commande qu'il référence, la liste des mots clés actifs et leurs valeurs, le type et le nom du concept produit.

La construction puis l'exécution de l'objet JDC déclenchent les actions suivantes :

- analyse syntaxique du fichier de commande utilisateur : c'est à ce niveau que la syntaxe python est vérifiée (parenthèses, virgules entre mots clés, indentation …). La détection d'une erreur (`Syntax Error Python`) provoque l'arrêt de l'exécution d'Aster. La première erreur est fatale : on ne recherche pas les erreurs suivantes,
- construction des étapes : cela consiste à créer un objet ETAPE pour chaque appel à une commande Aster dans le fichier de commandes. Cet objet est enregistré auprès de JDC qui gère la liste des étapes et des concepts afférents,
- vérification de chaque ETAPE : si l'appel à une commande dans le fichier utilisateur est incohérent avec le catalogue de commandes, un compte rendu est affiché et l'exécution est arrêtée à ce niveau. C'est la vérification sémantique,
- exécution proprement dite des commandes : pour chaque étape prise dans l'ordre, appel à la routine FORTRAN de haut niveau (op0nnn.f) correspondante.

2.2 L'exécution globale ou pas à pas

Un jeu de commande peut être construit et exécuté selon deux modes :

- le mode global pour lequel **toutes** les étapes du jeu de commande sont d'abord construites puis exécutées dans leur ordre d'apparition. Ce mode est choisi par le mot clé `PAR_LOT='OUI'` dans la commande de démarrage `DEBUT`,
- le mode pas à pas pour lequel chaque étape est immédiatement exécutée après sa construction. Ce mode est choisi par le mot clé `PAR_LOT='NON'` dans la commande `DEBUT`.

Si l'utilisateur ne précise rien dans la commande de démarrage, le mode global (PAR_LOT='OUI') est retenu. Ces deux modes présentent chacun leurs avantages et inconvénients.

Le mode d'exécution global garantit à l'utilisateur que tout son fichier est sémantiquement correct avant d'entamer des calculs qui pourraient échouer ou ne pas converger. Il serait en effet dommage de s'arrêter en erreur fatale après une longue résolution à cause d'un mot clé oublié dans une commande de post-traitement.

Le mode pas à pas ne construit une étape qu'après avoir exécuté la précédente. Il ne détecte donc que les erreurs sémantiques de la commande en cours et présente l'inconvénient décrit ci-dessus. Il permet cependant d'exploiter un résultat calculé (dans un concept) dans le fichier de commande pour, par exemple, y placer des instructions conditionnelles.

Voici un exemple de boucle avec un critère d'arrêt sur la valeur d'une grandeur calculée, stockée dans le concept de type `table` : `RELV[k]`. Si par exemple un mot clé obligatoire manque dans l'appel à `POST_RELEVE_T`, cela ne sera détecté qu'après l'exécution complète du premier `MECA_STATIQUE`. Par contre, le mode pas à pas rend ici possible l'affectation de la variable `SYY` puisque le concept `RELV[k]` a été complètement calculé au moment où le superviseur exécute cette ligne.

```
DEBUT(PAR_LOT='NON')

RESU=[None]*10
RELV=[None]*10

for k in range(1,10):

   RESU[k]=MECA_STATIQUE( ... )
   RELV[k]=POST_RELEVE_T( ... )
   SYY=RELV[k]['VMIS',4]

   if SYY < critere :
     break

FIN()
```

Il faut noter que le choix d'un mode d'exécution conditionne l'ordre dans lequel se déroulera l'analyse **sémantique** (ETAPE par ETAPE ou globalement pour tout le JDC). Mais, dans les deux cas, l'analyse **syntaxique** python est toujours faite au préalable pour tout le fichier de commande.

Remarque :

EFICAS ne peut générer et relire que des jeux de commandes contenant exclusivement des commandes ASTER, sans autres instructions python ; ceci indépendamment du mode `PAR_LOT` choisi.

2.3 La construction des étapes

Lors de la construction de chaque objet ETAPE, on vérifie sa cohérence sémantique avec le catalogue de la commande à laquelle il se réfère. Toute erreur détectée est consignée dans un compte rendu qui, en mode d'exécution globale, est délivré après l'analyse de tout le fichier de commande.

Exemples de vérifications sémantiques :

- respect du nombre d'arguments des mots clés,
- respect du type d'argument,
- appartenance d'un argument à une liste de valeurs possibles,
- exactitude de l'orthographe d'un mot clé ou d'un mot clé facteur,
- respect des règles d'exclusion ou d'implication entre mots clés,
- présence des mots clés obligatoires.

A ce stade, si la commande est un opérateur et produit un concept, celui-ci est typé. Le superviseur vérifie qu'un concept de même nom n'a pas été déjà défini, ou s'il est réemployé, que la commande l'autorise.

2.4 Le traitement des macro-commandes

Une macro-commande, vu l'utilisateur, est une commande ordinaire. En fait, elle n'appelle pas directement une routine Fortran de haut niveau mais génère d'autres commandes.

Deux types de macro-commandes existent :

- les macros en Python,
- les macros superviseur : ce sont les commandes spéciales (`DEBUT`, `FORMULE`, `INCLUDE`, `INCLUDE_MATERIAU`, `POURSUITE`) qui nécessitent un traitement au niveau de leur construction.

Au même titre que le JDC lui-même, l'appel à une macro-commande produit un objet père (de type MACRO-ETAPE) qui contient des objets fils : les étapes que génère la macro, voire d'autres macros.

Une macro-commande du JDC est tout d'abord traitée comme les autres commandes (vérification syntaxique, construction de la macro étape). Puis elle est «construite » par application de la méthode python `Build` sur l'objet JDC. Après sa construction, les étapes des commandes produites par la macro sont substituées à l'étape de la macro elle-même, pour exécution ultérieure.

Il est important de noter que la phase de construction des macro-commandes se déroule juste avant leur exécution, et non lors la passe globale sur le fichier de commande en mode `PAR_LOT='OUI'`. Cela a deux conséquences :

- EFICAS analyse la syntaxe de la macro-commande elle-même, mais pas celle de ses sous-commandes.
- On peut en revanche exploiter, dans la programmation des macros, des données précédemment calculées et rapatriées dans l'espace de noms python, sans avoir à imposer le mode `PAR_LOT='NON'` à l'utilisateur de la macro.

2.5 Les procédures de démarrage

Les procédures de démarrage disponibles sont :

`DEBUT` (cf. [U4.11.01] et `POURSUITE` [U4.11.03])

Au moins une de ces deux procédures doit être obligatoirement présente dans le fichier de commande. Aucune autre commande Aster ne doit les précéder. Si c'est le cas ou si aucune n'est présente, l'exécution sera arrêtée dès la création du JDC. Ce sont ces procédures qui contiennent l'information sur le mode d'exécution (`PAR_LOT='OUI'` ou `'NON'`) qui conditionne le mode d'exécution des commandes qui suivent.

Ce sont des macro-commandes superviseur qui, à leur construction, appellent les routines Fortran permettant d'initialiser le calcul par les tâches suivantes :

- "connexion" des unités logiques des fichiers standards,
- ouverture des bases de données,
- lecture du catalogue d'éléments.

La première tâche consiste à mettre en correspondance des numéros d'unités logiques de fichiers d'entrée/sortie standards (message, erreur, résultat).

La deuxième tâche consiste à définir et ouvrir les bases de données (fichier d'accès direct utilisés par le gestionnaire de mémoire) conformément aux instructions de l'utilisateur, qui peut redéfinir des paramètres de ces fichiers (voir documents [U4.11.01] et [U4.11.03] sur les procédures de démarrage). On appelle pour cela les routines d'initialisation `JEVEUX` (voir document [D6.02.01] le Gestionnaire de mémoire, `JEVEUX`).

La séquence des commandes à exécuter se termine obligatoirement par la commande `FIN`. Le texte qui suit `FIN` doit être commentarisé (c'est-à-dire débuter par #). Pour un fichier inclus, c'est la commande `RETOUR` qui marque la fin des instructions qu'ASTER doit prendre en compte.

Remarque :

> *En mode interactif, saisie des commandes à la main, ne pas mettre de commande* `FIN` *et passer l'argument* `-interact` *sur la ligne de commande de soumission du job.*

2.6 Liens avec EFICAS

Le noyau du superviseur est commun avec Eficas, l'éditeur de fichiers de commandes Aster. Lors de l'édition d'un fichier de commande, celui-ci réalise l'analyse syntaxique et les vérifications de cohérence des concepts par construction du JDC et de ses objets ETAPE. Eficas ne réalise bien sûr pas la tâche de construction des macro-commandes qui nécessiterait le code source d'*Aster*.

3 Le langage de commande

3.1 Python et le langage de commande

Un fichier de commandes pour le *Code_Aster* est exclusivement composé d'instructions Python. La première des contraintes est donc de se conformer aux règles de ce langage. On pourra lire le tutoriel Python (www.python.org) ou les nombreux livres d'introduction à Python pour plus de détail, mais ce n'est pas nécessaire pour l'usage d'Aster.

Un fichier de commandes peut contenir des instructions python de deux natures : des commandes *Aster* et … n'importe quelle autre instruction python. En effet, un fichier de commandes est un programme python à part entière et on peut en particulier y placer des structures de contrôle (boucles), des tests (if), des calculs numériques, des appels à des fonctions de pré et post-traitement.

Dans le cadre d'une utilisation « classique » du code où le fichier de commande contient exclusivement des commandes Aster, les deux règles spécifiques à Python à retenir sont :

- Pas d'indentation sur la première ligne de déclaration d'une instruction.
  ```
  mail = LIRE_MAILLAGE ( )
  ```
  Il ne faut placer ni blanc, ni tabulation avant la chaîne de caractères `mail`.

- Les arguments des fonctions, autrement dit les mots clés des commandes, sont séparés par des virgules ; ils se composent d'un mot clé, du signe « = », du contenu du mot clé.

Important :

L'éditeur EFICAS ne permet de produire que des fichiers de commandes de ce type : contenant exclusivement des commandes ASTER, sans autre instruction Python. Utiliser EFICAS garantit essentiellement trois choses :

- le fichier produit aura une syntaxe python correcte,
- les commandes produites seront cohérentes avec le catalogue de commandes,
- les concepts produits seront correctement enchaînés (pas d'utilisation d'un concept sans qu'il ait été créé par une commande précédente).

L'utilisateur ayant composé ainsi son fichier de commande sera à l'abri d'un arrêt à l'exécution au motif d'un problème de syntaxe.

3.2 Notion de concept

Définition : on appelle *concept* les structures de données Aster, que l'utilisateur peut manipuler et nommer. Ces concepts sont typés au moment de leur création et ne pourront être utilisés que comme argument d'entrée du type correspondant dans une commande ultérieure.

La notion de concept permet donc à l'utilisateur de manipuler des objets symboliquement et indépendamment de leur représentation interne (qu'il peut ne pas connaître). D'ailleurs, l'objet python désigné par le nom du concept ne contient aucune autre information que son type, sa classe au sens python (cf. doc D). Son nom, transmis par le superviseur au FORTRAN, permet à Aster de retrouver la structure de données correspondante dans la base globale. Mais il n'est pas possible d'avoir visibilité de la structure de données depuis le fichier de commande. Par exemple, les instructions suivantes ne permettent pas d'imprimer la structure de données de type `maillage` et de nom `mail` :

```
mail=LIRE_MAILLAGE ( )
print mail
```

mais génèrent le message suivant :

```
SD maillage nom : mail
```

Il y a une exception à cette règle : les tables. En effet, un artifice de programmation permet de récupérer simplement de l'information contenue dans une structure de donnée `TABLE` en manipulant celle-ci comme un tableau à deux entrées :

pour imprimer toute la table : `print resu`
pour imprimer une valeur : `print resu[ 'DX' , 1]`
pour l'affecter à une variable : `valeur = resu[ 'DX' , 1]`

Cela suppose bien sûr que la structure de données `resu`, de type `TABLE`, a déjà été calculée au moment où on rencontre cette instruction : donc en mode d'exécution pas à pas (`PAR_LOT='NON'`).

Remarque lexicale :

> *Les noms de concepts ne doivent pas dépasser 8 caractères. Les caractères alphanumériques sont licites (lettres minuscules et majuscules et chiffres non placés en première position) ainsi que le underscore '_'. La casse est importante : les concepts « MAIL » et « Mail » pourront être utilisés dans un même fichier de commande et seront considérés comme différents … c'est toutefois déconseillé pour la lisibilité du fichier !*

3.3 Opérations possibles

La structure du langage de commande se présente sous la forme d'une suite linéaire d'instructions. Outre les instructions python autres que des commandes Aster, dont il n'est pas question pour le moment, trois natures d'instructions (ou de commandes) sont disponibles :

- l'opérateur qui effectue une action et qui fournit un `concept produit` d'un type prédéfini exploitable par les instructions suivantes dans le jeu de commandes,
- la procédure qui effectue une action mais ne fournit pas de concept,
- la macro-commande qui génère une séquence d’instructions des deux types précédents et qui peut produire zéro, un ou plusieurs concepts.

Typiquement, un opérateur sera une commande d’affectation ou de résolution, une procédure sera une commande d'impression (dans un fichier).

Du point de vue syntaxique un opérateur se présente sous la forme :

```
nomconcept = opérateur( arguments . . . )
```

Alors qu'une procédure se présente sous la forme :

```
procédure( arguments . . . )
```

La syntaxe d'un opérateur ou d'une procédure est décrite dans le paragraphe suivant.

3.4 Règles sur le concept produit par un opérateur

3.4.1 Principe de base

A chaque exécution d'un opérateur, celui-ci fournit un nouveau concept produit du type prédéfini dans le catalogue de commande.

Les concepts apparaissant en argument d'entrée des commandes, ne sont pas modifiés.

3.4.2 Concept produit et concept réutilisé

On appelle concept réutilisé, un concept qui étant produit par un opérateur, est modifié par une nouvelle occurrence de cet opérateur ou par un autre opérateur.
L'utilisation d'un concept réutilisé n'est possible, comme dérogation du Principe de Base qu'à deux conditions :

- autorisation donnée, par le catalogue et la programmation de la commande, d'utiliser des concepts réutilisables pour l'opérateur : l'attribut `reentrant` du catalogue vaut `'o'` ou `'f'`,
- demande explicite de l'utilisateur de la réutilisation d'un concept produit par l'attribut `reuse=nom_du_concept` dans les arguments des commandes qui le permettent.

3.4.3 Vérifications effectuées par le superviseur sur les concepts produit

- Concept produit respectant le principe de base :
  Le superviseur vérifie que le nom du concept produit n'est pas déjà attribué par une des commandes précédentes, en particulier par une commande d'une exécution précédente dans le cas d'une `POURSUITE` ou d'un `INCLUDE`.
- Concept utilisé en réutilisation :
  Le superviseur vérifie que :
  - le nom du concept produit est déjà bien attribué.
  - l'opérateur est bien habilité à accepter des concepts réutilisés,
  - le type du concept est conforme au type de `concept produit` par l'opérateur.

Exemples commentés :

```
DEBUT ( )
concept=opérateur ( )               #  (1)
```
est correct : on défini le concept,
```
concept=opérateur ( )               #  (2)
```
est incorrect : on essaie de redéfinir le

```
                                          #   concept mais sans le dire,
concept=opérateur (reuse = concept )      #   (3)   est correct, si l'opérateur accepte des
                                          #   concepts existants et si le type est
                                          #   cohérent ; c'est incorrect si l'opérateur
                                          #   ne les accepte pas.
FIN ( )
```

De fait un concept ne peut être créé qu'une seule fois : ce qui signifie apparaître à gauche du signe = (égal) sans que `reuse` soit employé dans les arguments de la commande.

Dans le cas d'une ré-utilisation, préciser à nouveau le nom du concept derrière l'attribut `reuse` est redondant ; d'autant plus que le superviseur vérifie que les deux noms de concept sont identiques.

Remarque :

On peut détruire un concept, et donc ré-utiliser son nom ensuite.

3.5 Corps d'une commande

3.5.1 Introduction

Le corps d'une commande contient la partie "variable" de la commande. Les déclarations sont séparées par des virgules et à part l'attribut `reuse` mentionné plus haut, elles sont toutes de la forme :

`[mot_clé] = [argument]`

Le mot clé est nécessairement un mot clé de la commande en cours, déclaré dans le catalogue de celle-ci.

3.5.2 Mot clé

Un mot clé est un identificateur formel, c'est le nom de l'attribut recevant l'argument.

Exemple : `MATRICE` = ...

Remarques syntaxiques :

- *l'ordre d'apparition des mots clés est libre, il n'est pas imposé par l'ordre de déclaration dans les catalogues,*
- *les mots clés ne peuvent excéder 16 caractères (mais seuls les 10 premiers caractères sont signifiants).*

Il existe deux types de mots clés : les mots clés simples et les mots clés facteurs qui diffèrent par la nature de leurs arguments.

3.5.3 Argument d'un mot clé simple

3.5.3.1 Le type des arguments

Les types de base reconnus par le superviseur sont :

- les entiers,
- les réels,
- les complexes,
- les textes,
- les logiques,
- les concepts,
- ainsi que les listes de ces types de bases.

Les entiers et les réels correspondent exactement aux types équivalents en python.

- Mot clé simple facultatif attendant un réel :
  Catalogue : `VALE = SIMP( statut='f' , typ = 'R' ) ,`
  Fichier de commandes : `VALE = 10. ,`
- Mot clé simple facultatif attendant un entier :
  Catalogue : `INFO = SIMP( statut='f' , typ = 'I' ) ,`
  Fichier de commandes : `INFO = 1 ,`

La représentation du type complexe est un "tuple" python contenant une chaîne de caractères indiquant le mode de représentation du nombre complexe (parties réelle et imaginaire ou module-phase) puis les valeurs numériques.

Catalogue : `VALE_C = SIMP( statut='f' , typ = 'C' ) ,`
Fichier de commandes : `VALE_C = ( 'RI' , 0.732 , -0.732 ) ,`
Fichier de commandes : `VALE_C = ( 'MP' , 1. , -45. ) ,`

Les deux notations sont strictement équivalentes. En notation `'MP'`, la phase est en degrés.

Le type texte est déclaré entre simples cotes. La casse est respectée. Cependant, quand un mot clé doit prendre une valeur dans une liste prédéfinie dans le catalogue, l'usage veut que cette valeur soit aujourd'hui toujours en capitales.

Catalogue : `TOUT =SIMP( typ='TXM',into=('OUI','NON')) ,`
Fichier de commandes : `TOUT = 'OUI' ,`

La casse est importante et, dans le contexte ci-dessus, la ligne de commande suivante échouera :

Fichier de commandes : `TOUT = 'oui' ,`

Le type logique n'est pas utilisé aujourd'hui dans le catalogue de commandes.

Le concept est déclaré simplement par son nom, sans cotes ni guillemets.

3.5.3.2 Notion de liste

Attention :

Le mot « liste » est ici un abus de langage. Il ne s'agit pas du type « liste » de python mais plutôt de tuples, au sens de python : les différents items sont déclarés entre une parenthèse ouvrante et une parenthèse fermante ; ils sont séparés par des virgules.

Les listes sont des listes homogènes, c'est à dire dont les éléments sont du même type de base. Tout type de base peut être utilisé en liste.

Exemples de liste :

liste d'entiers `(1, 2, 3, 4),`
liste de texte `( 'ceci', 'est', 'une', 'liste', 'de', 'texte'),`
liste de concepts `( resu1, resu2, resu3 ),`

Facilité d'emploi :

Il est admis qu'une liste réduite à un élément puisse être décrite sans parenthèse.

Exemple de liste erronée :

Liste hétérogène d'entier et de réel

```
(1, 3, 4.)
```

3.5.4 Mot clé facteur

Certaines informations ne peuvent être données globalement (en une fois dans la commande), il est donc important de prévoir la répétition de certains mots clé, pour pouvoir leur affecter des arguments différents. Le mot clé facteur offre cette possibilité ; sous un mot clé facteur, on trouvera donc un ensemble de mots clé (simples), qui pourront être utilisés à chaque occurrence du mot clé facteur. Cela permet en outre d’améliorer la lisibilité du fichier de commandes en regroupant des mots clés qui partagent un sens commun : par exemple les différents paramètres d’un même matériau.

Contrairement au mot clé simple, le mot clé facteur ne peut recevoir qu'un seul type d'objet : l'objet superviseur « _F », ou une liste de celui-ci.

Soit le mot clé facteur n'a qu'une seule occurrence et on peut écrire par exemple, au choix :

```
IMPRESSION =     _F(   RESULTAT = resu, UNITE = 6),
```
ou
```
IMPRESSION =  (  _F(   RESULTAT = resu, UNITE = 6), ),
```

Dans le premier cas, le mot clé facteur `IMPRESSION` reçoit un objet `_F`, dans l'autre, il reçoit un singleton. Attention à la virgule ; en python, un tuple à un élément s'écrit : `(élément,)`

Soit le mot clé facteur a plusieurs occurrences, deux dans cet exemple :

```
IMPRESSION =  (  _F(   RESULTAT = resu1,   UNITE = 6),
                 _F(   RESULTAT = resu2,   UNITE = 7) ),
```

Le nombre d'occurrence (minimum et/ou maximum) attendu d'un mot clé facteur est défini dans le catalogue de commandes.

Notion de valeur par défaut

Il est possible de faire affecter par le superviseur des valeurs par défaut. Ces valeurs sont définies dans le catalogue de commandes et non dans le `FORTRAN`.

Il n'y a pas de distinction du point de vue de la routine associée à la commande entre une valeur fournie par l'utilisateur et une valeur par défaut introduite par le superviseur. Ceci apparaît lors de l'impression des commandes utilisateur par le superviseur dans le fichier de messages : toutes les valeurs par défaut apparaissent dans le texte de commande, si elles n'ont pas été fournies par l'utilisateur

Rappel : on ne peut pas donner de valeur par défaut à un concept.

4 Définition de valeurs et évaluations d'expressions

Il est possible d'affecter des valeurs à des variables python afin d'utiliser celles-ci comme arguments de mots-clés simples : ces variables sont appelées paramètres dans EFICAS. Elles peuvent contenir des valeurs entières, réelles, complexes, des textes ou des listes de ces types.

Exemple :

```
young =  2.E+11
mat      =  DEFI_MATERIAU ( ELAS = _F(  E = young , NU = 0.3 ) )
```

En fin d'exécution, le contexte python est sauvegardé avec la base. Ainsi, dans la poursuite qui suivra, les paramètres seront toujours présents, avec leurs valeurs prédéfinies, tout comme les concepts ASTER.

Il est possible de réaliser des opérations en python sur les arguments de mots-clés simples :

```
Pisur2   =  pi/2.
mat      =  MA_COMMANDE( VALE = Pisur2 )
```

ou :

```
var      =  ' world'
mat      =  MA_COMMANDE(  VALE  = pi/2. ,
                          VALE2 = Pisur2+cos(30.) ,
                          TEXTE = 'hello'+var    )
```

5 Usage de python dans les fichiers de commande

Il n'est pas nécessaire de connaître le langage python pour utiliser *Code_Aster*. En effet, moyennant quelques règles de base à respecter sur l'indentation et le parenthésage, seule la connaissance du langage de commande décrit dans les catalogues de commande est nécessaire. Et encore, EFICAS permet de se dispenser de recourir au catalogue ou au paragraphe « syntaxe » des commandes en proposant graphiquement les mots clés à renseigner.

Toutefois, l'utilisateur avancé pourra utiliser à bon compte la puissance du langage PYTHON dans son fichier de commandes, puisque celui-ci est déjà écrit dans ce langage.

Les quatre usages principaux peuvent être : l'écriture de macro-commandes personnalisées, l'usage d'instructions générales python, l'import de modules python utiles, la récupération d'information des structures de données *Code_Aster* dans des variables PYTHON.

Remarque :

Si on veut utiliser des caractères français accentués dans le fichier de commandes ou les modules importés, il faut placer l'instruction suivante en première ou deuxième ligne du fichier :

```
# -*- coding: iso-8859-1 -*-
```

En python 2.3, l'absence de cette ligne provoque un warning qui deviendra une erreur en python 2.4 ; dans ASTER, c'est systématiquement une erreur.

5.1 Macro-commandes personnalisées

Voir le document [D5.01.02] : Introduire une nouvelle macro-commande
Les macro-commandes personnalisées sont très faciles à programmer. Elles peuvent servir à capitaliser des schémas de calcul récurrents et ainsi constituer un outil-métier. Il est fortement conseillé de prendre exemple sur les macro-commandes existantes : package Macro dans bibpyt.

5.2 Instructions générales PYTHON et modules utiles

Les utilisateurs avancés peuvent tirer grand profit de l'emploi de boucles (for), de tests (if), des exceptions (try, except) et de manière générale de toute la puissance du langage PYTHON directement dans leur fichier de commandes. La liste des usages est impossible à établir exhaustivement. De nombreux exemples sont présents dans les cas tests de la base de tests. On peut par exemple faire de l'adaptation de maillage en plaçant la séquence calcul/remaillage dans une boucle, établir un critère d'arrêt des itérations par un test sur une valeur calculée.

Consulter le paragraphe suivant dédié aux exceptions Aster « particularisées ».

Dans une boucle, si on recrée un concept déjà existant, il faut penser à le détruire au préalable par la commande `DETRUIRE`.

Les autres fonctionnalités diverses de python intéressantes pour l'utilisateur de *Code_Aster* peuvent être :

- la lecture-écriture sur fichier,
- le calcul numérique (par exemple en utilisant Numerical Python),
- l'appel via le module `os` au langage de script, et en particulier le lancement d'un code tiers (`os.system`)
- la manipulation de chaînes de caractères
- l'appel à des modules graphiques (grace, gnuplot)

5.3 Exceptions PYTHON du module aster

Le mécanisme des exceptions Python est très intéressant, il autorise par exemple à « essayer » une commande puis reprendre la main si celle-ci « plante » en levant une exception particulière :

```
try:
    bloc d'instructions
except ErreurIdentifiée, message:
    bloc d'instructions exécuté si ErreurIdentifiée se
produit…
```

Dans le fichier de commandes, on peut utiliser ce mécanisme avec n'importe quelle exception des modules Python standards.
On dispose également d'exceptions propres à *Code_Aster* (au module `aster`), réparties en deux catégories, les exceptions associées aux erreurs <S>, et celle associée aux erreurs <F>.

5.3.1 Interception des erreurs <S>

En cas d'erreur <S>, l'erreur est identifiée par une de ces exceptions :

- `aster.NonConvergenceError` en cas de non convergence du calcul,
- `aster.EchecComportementError` problème lors de l'intégration de la loi de comportement,
- `aster.BandeFrequenceVideError` pas de mode trouvé dans la bande de fréquence,
- `aster.MatriceSinguliereError` matrice singulière,
- `aster.TraitementContactError` problème lors du traitement du contact,
- `aster.MatriceContactSinguliereError` matrice de contact singulière,
- `aster.ArretCPUError` arrêt par manque de temps CPU.

Toutes ces exceptions dérivent de l'exception <S> générale qui est `aster.error`. Ce qui signifie que `except aster.error` intercepte toutes les exceptions citées ci-dessous. Il est néanmoins toujours préférable de préciser à quelle erreur on s'attend !

Exemple d'utilisation :

```
try:
    resu = STAT_NON_LINE(...)
except aster.NonConvergenceError, message:
    print 'Arrêt pour cette raison : %s' % str(message)
    # On sait qu'il faut beaucoup plus d'itérations pour
converger
    print "On continue en augmentant le nombre
d'itérations."
    resu = STAT_NON_LINE(reuse = resu,
                         ...,
                         CONVERGENCE=_F(ITER_GLOB_MAXI=400,)
,)
except aster.error, message:
```

```
    print "Une autre erreur s'est produite : %s" %
str(message)
    print "Stop"
    from Utilitai.Utmess import UTMESS
    UTMESS('F', 'Exemple', 'Erreur <S> inattendue')
```

5.3.2 Interception des erreurs <F>atales

En cas d'erreur fatale, le comportement est modifiable par l'utilisateur.
Par défaut, le code s'arrête en erreur <F>, on peut voir la remontée d'erreur (appel des routines fortran) dans le fichier de sortie.
Si on le souhaite, le code peut lever l'exception `aster.FatalError`, et dans ce cas (comme pour une erreur <S>), si l'exception est interceptée par un `except`, l'utilisateur reprend la main, sinon le code s'arrête (pas de remontée d'erreur fortran).
Ce choix est déterminé dans les commandes `DEBUT/POURSUITE`, mot-clé facteur `ERREUR` (cf. [U4.11.01] et [U4.11.03]) et à tout moment par la méthode `aster.onFatalError`.

Cette dernière appelée sans argument retourne le comportement actuel en cas d'erreur fatale :

```
comport = aster.onFatalError()
print "Comportement courant en cas d'erreur fatale : %s" % comport
```

et permet de définir le comportement que l'on souhaite :

```
aster.onFatalError('EXCEPTION')     # on lève l'exception FatalError
```

ou

```
aster.onFatalError('ABORT')         # on s'arrête avec remontée d'erreur.
```

5.3.3 Validité des concepts en cas de levée d'exception

Lors qu'une exception est levée et interceptée (par `except`), le concept produit par la commande dans laquelle l'erreur s'est produite est rendu tel quel, la commande n'ayant pas terminé normalement.
Dans certains cas, notamment après une erreur fatale, il se peut que le concept produit ne soit pas utilisable ; utiliser alors `DETRUIRE` pour le supprimer.
De même, si l'on souhaite réutiliser le nom du concept pour en créer un nouveau, il faut `DETRUIRE` celui obtenu dans le `try`.

En cas d'erreur <S> levées dans `STAT/DYNA_NON_LINE`, le concept est généralement valide, et peut être réutilisé (mot-clé `reuse`) pour poursuivre le calcul avec une autre stratégie (comme dans l'exemple cité précédemment).

Enfin, avant de rendre la main à l'utilisateur, les objets de la base volatile sont supprimés par un appel à `JEDETV`, et la marque Jeveux est repositionnée à 1 (avec `JEDEMA`) pour libérer les objets ramenés en mémoire.

5.3.4 Précautions d'emploi des exceptions

- Les deux exceptions `aster.error` et `aster.FatalError` sont indépendantes (aucune ne dérive de l'autre), ce qui signifie que si on souhaite reprendre la main en cas d'erreur <S> et d'erreur <F> :

  1) il faut activer la levée d'exception en cas d'erreur <F> avec `aster.FatalError('EXCEPTION')` ou dans `DEBUT/POURSUITE`.
  2) il faut intercepter les deux exceptions :
     `except (aster.FatalError, aster.error), message:  …`

- Il est déconseillé d'utiliser « `except:` » sans préciser à quelle exception on s'attend (c'est une règle générale en Python indépendamment des exceptions Aster). En effet, le traitement effectué sous l'`except` a peu de chance d'être valable dans tous les cas d'erreur.
  De même, comme dans l'exemple donné plus haut, il est préférable d'utiliser les exceptions particularisées `NonConvergenceError`, `ArretCPUError`, … que l'exception de plus haut niveau `aster.error` ; toujours dans l'idée de savoir exactement ce qui s'est produit.

5.4 Récupération de valeurs calculées dans des variables PYTHON

Exploiter le langage PYTHON dans son fichier de commande n'est intéressant que si on peut conditionnellement lancer des actions en fonction de ce que le code a calculé.

Certaines passerelles existent entre python et les structures de données calculées par le code et présentes dans la mémoire JEVEUX. D'autres restent à programmer ; ceci est un domaine en évolution et de futurs développements sont attendus.

Il est essentiel de comprendre que récupérer des données calculées nécessite que les instructions entraînant leur obtention aient bien été exécutées au préalable. Autrement dit, il est indispensable d'exécuter le code en mode `PAR_LOT='NON'` (mot clé de la commande `DEBUT`). En effet, dans ce cas, il n'y a pas d'analyse globale du fichier de commande, mais chaque instruction est exécutée séquentiellement. Quand on arrive sur une instruction, tous les concepts la précédent ont donc déjà été calculés.

Voici quelques méthodes d'accès aux structures de donnée. La liste est non exhaustive, se reporter à la documentation [U1.03.02].

Structure de données	**Méthode**	**Type python retourné**	**Information retournée**
listr8	LIST_VALEURS	liste	Liste des valeurs
maillage	LIST_GROUP_NO	liste	Liste des groupes de nœuds
	LIST_GROUP_MA	liste	Liste des groupes de mailles
table	[…]	réel	Contenu de la table
fonction	LISTE_VALEURS	liste	Liste des valeurs
résultat	LIST_CHAMPS	liste	Liste des champs calculés
	LIST_NOM_CMP	liste	Liste des composantes
	LIST_VARI_ACCES	liste	Liste des variables d'accès
	LIST_PARA	liste	Liste des paramètres
cham_no	EXTR_COMP	post_comp_cham_no	Contenu du champ dans une table
cham_elem	EXTR_COMP	post_comp_cham_el	Contenu du champ dans une table
Tout objet JEVEUX	getvectjev	liste	Liste des objets du vecteur jeveux